Scientific Programming 19 (2011) 199–212
DOI 10.3233/SPR-2011-0328
IOS Press

Use of GPU computing for uncertainty quantification in computational mechanics: A case study

Gaurav and Steven F. Wojtkiewicz *
Department of Civil Engineering, University of Minnesota, Minneapolis, MN, USA

Abstract. Graphics processing units (GPUs) are rapidly emerging as a more economical and highly competitive alternative to CPU-based parallel computing. As the degree of software control of GPUs has increased, many researchers have explored their use in non-gaming applications. Recent studies have shown that GPUs consistently outperform their best corresponding CPU-based parallel computing alternatives in single-instruction multiple-data (SIMD) strategies. This study explores the use of GPUs for uncertainty quantification in computational mechanics. Five types of analysis procedures that are frequently utilized for uncertainty quantification of mechanical and dynamical systems have been considered and their GPU implementations have been developed. The numerical examples presented in this study show that considerable gains in computational efficiency can be obtained for these procedures. It is expected that the GPU implementations presented in this study will serve as initial bases for further developments in the use of GPUs in the field of uncertainty quantification and will (i) aid the understanding of the performance constraints on the relevant GPU kernels and (ii) provide some guidance regarding the computational and the data structures to be utilized in these novel GPU implementations.

Keywords: Uncertainty quantification, GPU computing, parallel computing, random vibration, computational mechanics

1. Introduction

Primarily driven by the gaming industry, GPUs have evolved from being partially programmable in 1999 to a much more economical and highly competitive alternative architecture to CPU-based parallel computing in 2011. As the degree of software control (programmability) of GPUs has increased, its use in the solution of general purpose computing problems has spawned the discipline of general purpose GPU (GPGPU) computing. The interest of researchers in GPGPU is primarily due to four reasons: (i) better performance, (ii) anticipated evolution, (iii) cost efficiency and (iv) energy efficiency of GPUs.

The performance of a given algorithm on a given processor is governed primarily by two factors: (i) peak computational capability of the processor (typically measured in giga floating point operations per second, Gflop/s) and (ii) memory bandwidth between the processor and its dynamic random access memory (typically measured in gigabytes per second, GB/s). Figures 1 and 2 show a comparison of these two performance metrics for GPUs vs contemporary CPUs. While the peak performance of current CPUs is less than 250 Gflop/s and 40 GB/s, GPUs have not only broken the teraflop/s barrier but also outperform CPUs in memory bandwidth by a factor of almost five. Apart from depicting a clear computational edge of GPUs compared to CPUs, Figs 1 and 2 highlight another differentiating feature of GPUs: their chronological development. While GPUs show an almost linear trend with time in their development, CPUs await a breakthrough in hardware development to take them out of the saturation point that started to be displayed in 2003.

The evolution gap between CPUs and GPUs can be attributed to their differing design philosophies. When CPUs stopped developing in accordance with the Moore's law [16] in 2003, multi-core CPUs were conceived. While multi-core chips provide exploitable parallelism to applications, their primary objective is to maintain, and possibly increase the performance of sequential programs. GPUs, on the other hand, were

* Address for correspondence: Steven F. Wojtkiewicz, Department of Civil Engineering, University of Minnesota, 240, Civil Engineering Building, 500 Pillsbury Dr SE, Minneapolis, MN 55455, USA. Tel.: +1 612 624 0063; Fax: +1 612 626 7750; E-mail: ykvich@umn.edu.

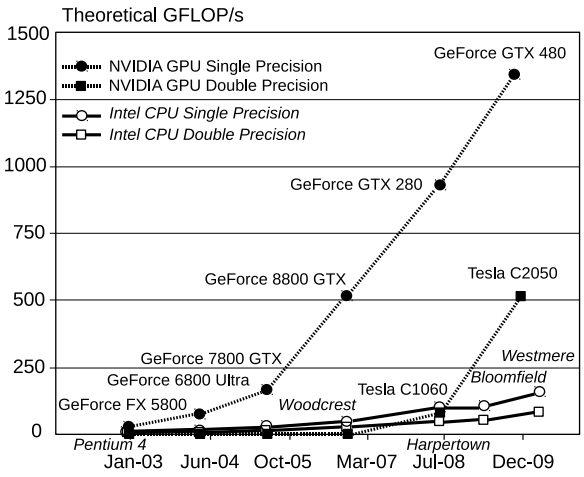


Fig. 1. Evolution of GPUs compared to CPUs (theoretical Gflop/s) [22].

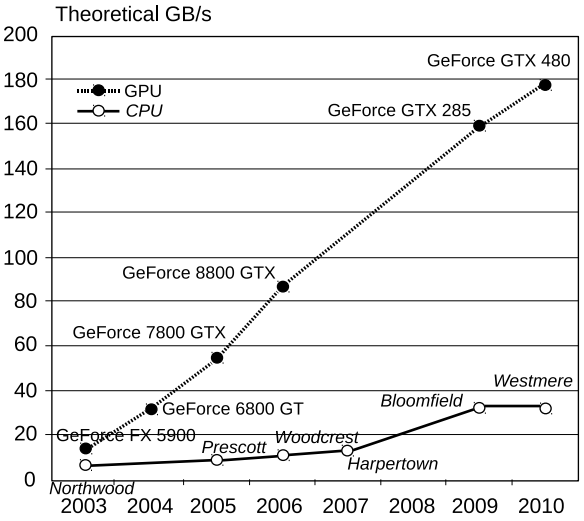


Fig. 2. Evolution of GPUs compared to CPUs (theoretical GB/s) [22].

initially designed to cater to computer games and graphics rendering, which require a large number of relatively lightweight computations. Thus, GPUs evolved based on the so-called many-core philosophy, which employs a large number of small computational cores on the same chip. The size of these cores allows GPUs to more economically possess a larger number of cores when compared to multi-core CPUs.

This economic viability along with the size of the cores on GPUs have a direct impact on their cost and energy efficiency. A comparison between Intel's Xeon E5530 quad-core processor [11] and NVIDIA's Tesla C1060 GPU [23] shows that the former yields a cost efficiency of about $4.5 per Gflop/s and an energy efficiency of about 0.7 watts per Gflop/s while the latter yields about $1.3 per Gflop/s and 0.2 watts per Gflop/s. These specific CPU and GPU models were chosen for the efficiency comparisons as this hardware has been utilized for performing the numerical studies presented later, herein. These models form a representative pair for comparing CPU and GPU performances due to their equivalence in market price and convenient availability in desktop computers.

GPUs are suitable for use in one of the very commonly employed parallelization strategies, single-instruction multiple-data (SIMD), because all the cores in each of their multiprocessors are single instruction multiple thread (SIMT) cores, i.e., all the cores execute the same set of instructions, but with different sets of data. A large number of processor cores (e.g., 240 in the NVIDIA Tesla C1060) along with a large number of active threads is the key to GPU's performance. Threads execute in groups, called warps, and the execution alternates between active warps with warps becoming temporarily inactive when waiting for data. This feature hides the memory latency to a great extent. Memory latency and the availability of only a small amount of shared memory among the threads are the two prime challenges for GPU algorithm developers. From a software point of view, programming on the GPUs has become considerably easier since the introduction of the NVIDIA CUDA™ programming environment, which is much like the programming language C.

Initially, GPGPU was primarily aimed towards the thirteen dwarfs or building blocks of parallel computing, seven of which were identified by Colella [6] while six others were added by Asanovic et al. [1]. These thirteen dwarfs, believed to be important for applications in science and engineering in at least the next decade, are listed in Table 1. Dense linear algebra and fast Fourier transform (FFT) algorithms have already matured into optimized routines and are available as vendor libraries for GPUs: CUBLAS [19] and CUFFT [20], respectively. CUBLAS implements the basic linear algebra subroutine (BLAS) equivalents and CUFFT implements one-, two- and three-dimensional fast Fourier transforms on GPUs. While these implementations are intended for pure GPU applications, hybrid algorithms have also been developed. Tomov et al. [29] have developed a suite of hybrid implementations of the equivalents of BLAS and LAPACK routines, which are available as a library, called MAGMA [14].

Table 1
Dwarfs of parallel computing

1. Dense linear algebra	6. Unstructured grids	11. Backtrack & branch + bound
2. Sparse linear algebra	7. Monte Carlo methods	12. Construct graphical models
3. Spectral methods (FFT)	8. Combinatorial logic	13. Finite state machine
4. N-body methods	9. Graph traversal	
5. Structured grids	10. Dynamic programming	

Bolz et al. [3] developed GPU implementations of conjugate gradient and multi-grid sparse matrix solvers. Later, Bell and Garland [2] and Vázquez et al. [31] developed efficient sparse matrix-vector product implementations. Tomov et al. [30] implemented probability-based simulations (Ising and percolation models) on GPUs. Recently, Tian and Benkrid [28] developed a GPU implementation of the Mersenne twister random number generation algorithm. Specific science and engineering applications can also be found in the literature. So [27] developed time-domain computational electromagnetics algorithms for GPU-based computers. Zhao [33] developed a GPU-accelerated partial differential equation (PDE) solver using the lattice Boltzmann model. Walsh et al. [32] developed GPU implementations of algorithms pertinent to geoscience and engineering system simulations. Januszewski and Kostur [13] developed a GPU solver for stochastic differential equations.

Keeping in mind that this paper is not a survey study of GPGPU applications, the literature survey has been limited to only the most pertinent studies within the context of uncertainty quantification problems. For the interested reader, detailed surveys and lists of the applications of GPGPU can be found elsewhere [4,5,9, 18,21,25,26].

The purpose of this study is to demonstrate the use of GPU computing in uncertainty quantification problems that arise in computational mechanics and dynamics. The field of uncertainty quantification addresses several types of analyses, such as uncertainty analysis/propagation, sensitivity analysis, model validation and calibration, and design exploration and optimization. A common theme among all these procedures is that they require reanalysis of similar systems. Hence, the computational techniques developed for one type of analysis can be easily adapted to another. The focus of the methods presented in this study concerns the uncertainty analysis of mechanical and dynamical systems.

A conceptual representation of the class of problems addressed in this study is shown in Fig. 3. $\mathcal{M}$ denotes an input/output model of a physical system of interest, which often consists of a large system of ordinary differential equations (ODEs), PDEs and/or algebraic equations; **f** denotes the input to the model and **y** denotes the set of desired outputs. Uncertainty can arise from primarily two sources: (i) the model itself and (ii) the input to the model. When a physical system cannot be deterministically modeled because of incomplete knowledge of its parameters (e.g., geometry, physical properties, etc.), it is required to either assume or experimentally deduce a statistical description of the same. The input to the system, which is usually a mathematical model of a physical process (e.g., earthquake, wind, etc.), is also often described most appropriately in a probabilistic formulation. Consequently, the outputs of interest do not remain deterministic, and it becomes necessary to (i) investigate and validate the probabilistic models of the system and the input and (ii) quantify the uncertainty of the system's response so that its long term behavior can be predicted with reasonable confidence.

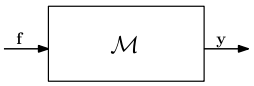


Fig. 3. Conceptual model.

Uncertainty analysis of such physical systems usually takes one of two paths. One entails discretization of the associated random fields/processes using, for instance, the Karhunen–Loeve expansion resulting in a set of PDEs with stochastic coefficients. Within the context of computational mechanics and dynamics, the stochastic finite element method (SFEM) [7] falls into this category. The other path consists of sampling-based methods. These include the Monte Carlo method [10], stratified sampling [17], Latin hypercube sampling [15], etc., and operate by first generating a large sample of the outputs of the uncertain system and then, computing the desired statistics. While the former class of methods usually shows faster convergence compared to the latter, it requires the knowledge of a probabilistic description of the associated uncertainties. In other words, SFEM-type methods work well with aleatory uncertainties but become problematic

when applied to epistemic uncertainties. Sampling-based methods, however, do not suffer from this limitation. As long as one can obtain the samples of the uncertain phenomena, either experimentally (e.g., seismogram records of earthquakes) or analytically, these methods can be utilized; the trade-off is the slow convergence rate of sampling-based methods.

However, with the advent of high-performance computing, sampling-based methods have maintained their feasibility as an attractive tool for uncertainty quantification studies. Moreover, known to be embarrassingly parallel, sampling-based methods are prime candidates for deploying parallel computing resources. Parallelism can be exploited in two ways depending on the size of the system in question. If the size of the system is small, the entire solution algorithm can be made to fit on one computational core, and different cores can be utilized to generate different samples of the desired response. When the size of the system is large, the solution procedure itself can be parallelized. These two strategies will be referred to as Type I and Type II parallelization strategies in the subsequent sections. The examples presented in this study are channelled towards sampling-based methods and utilize both of the aforementioned parallelization strategies.

Three implementation approaches have been utilized to deploy GPUs for uncertainty quantification problems in this study: (i) use of already existing, optimized GPU libraries, such as CUBLAS and CUFFT; (ii) development of new GPU kernels to address a certain analysis; (iii) mixed approach, which utilizes existing GPU libraries when possible and implements new GPU kernels otherwise. As will be demonstrated by the numerical examples, the first approach is straightforward and requires less work but also offers moderate improvements in computational efficiency. The second approach requires more work and a reformulation of many aspects of an algorithm, such as computational structure and data-structure, but considerable gains in the computational efficiency can be obtained. The third approach attempts to find a balance between the required effort and the gains obtained in computational efficiency. The numerical examples presented in this study demonstrate all the three approaches and also serve as intermediate layers, connecting the dwarfs of parallel computing to the uncertainty quantification problems, as depicted in Fig. 4. The procedures mentioned in Fig. 4 are not meant to be exhaustive but represent a large proportion of methods used in the uncertainty quantification of mechanical and dynamical systems.

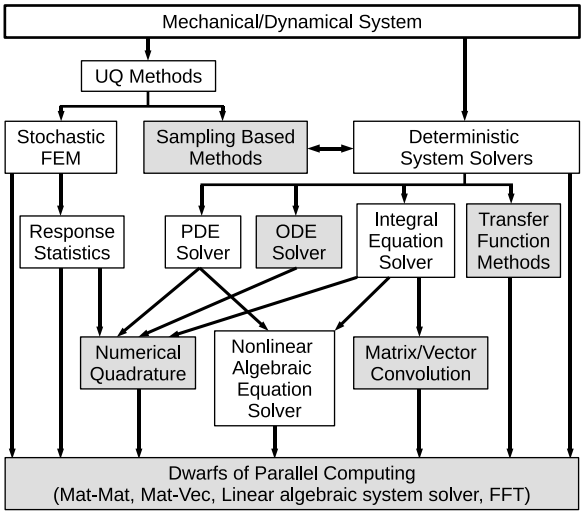


Fig. 4. Flow chart of primary and secondary building blocks.

These intermediate layers can be characterized into two categories. The first category consists of the methods that are utilized to analyze deterministic mechanical and dynamical systems. Time-domain analyses of dynamical systems are often posed as a system of PDEs, ODEs, integral equations or a combination of these. In the frequency domain, transfer function methods are employed. The response of mechanical systems to static loads can be determined by solving a system of algebraic equations involving the pertinent system matrices. The second category of intermediate layers consists of the uncertainty quantification methods as discussed earlier. It is evident that the parallelization of the first category is more apt for larger systems while that of the second category can be exploited for systems of all sizes.

Five types of analysis procedures, pertinent to the uncertainty quantification of computational mechanics models, depicted by shaded boxes in Fig. 4, have been implemented on the GPU. These are: (i) static response analysis of a structural system; (ii) frequency-domain analysis of a linear dynamical system; (iii) time-domain analysis of a linear dynamical system employing the convolution integral; (iv) direct Monte Carlo simulation of a small dynamical system in the time-domain using an explicit Rünge–Kutta time integration scheme; and (v) numerical quadrature of a function of random variables. As will be demonstrated by the numerical results, the GPU implementations of these analyses procedures offer considerable gains in computational efficiency compared to their corresponding multi-core CPU counterparts.

2. Theoretical background

This section develops the theoretical framework pertinent to the numerical examples presented in this study. Connections to the conceptual input/output model presented in Fig. 3 are also discussed.

2.1. Time-domain analysis of a dynamical system

A spatially discretized, n degree-of-freedom (DOF) dynamical system can be represented using a system of $2n$ ODEs in state-space as:

$$\dot{\mathbf{x}}(t) = \mathbf{A}(t)\mathbf{x}(t) + \mathbf{B}\mathbf{f}(t) + \mathbf{g}(\mathbf{x}), \tag{1}$$

where $\mathbf{A}(t)$ is the time-dependent system matrix, $\mathbf{B}$ is the load-distribution matrix, $\mathbf{g}(\mathbf{x})$ is a vector nonlinear function, $\mathbf{f}(t)$ is the external input, $\mathbf{x} = \{x_1, \ldots, x_n, \dot{x}_1, \ldots, \dot{x}_n\}^{\mathrm{T}}$ is the state-vector, and $\dot{(\cdot)}$ denotes derivative with respect to time. The response of such a system can be computed by utilizing a time-marching scheme, e.g., Rünge–Kutta methods.

For a linear, time-invariant system, the system of ODEs in (1) degenerates into a simpler form:

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{f}(t). \tag{2}$$

The response of the linear system in (2) can be written using the convolution integral as:

$$\mathbf{y}(t) = \mathbf{C}\left[\mathbf{x}_0 + \int_0^t \mathbf{h}(t-\tau)\mathbf{f}(\tau)\,\mathrm{d}\tau\right], \tag{3}$$

where $\mathbf{C}$ is an output matrix that maps the state-vector, $\mathbf{x}(t)$, to the desired output vector, $\mathbf{y}(t)$; $\mathbf{x}_0$ is the vector of initial conditions; and $\mathbf{h}(t)$ is the impulse response function of the system given by:

$$\mathbf{h}(t) = \mathrm{e}^{\mathbf{A}t}\mathbf{B}. \tag{4}$$

The model, $\mathcal{M}$, referred to in Fig. 3 is the physical system, represented via a system of ODEs, or an impulse response function arising from the ODE system; the input is the externally applied load; and the output is the desired response of the system.

2.2. Frequency-domain analysis of a dynamical system

Often, the input to the system, $\mathbf{f}(t)$, is random in nature and its description is most appropriately characterized in the frequency-domain by the power spectral density (PSD) of the random process governing the input. In such cases, if the system is linear time-invariant, the PSD of the response of the system can be directly computed using the following matrix triple-product:

$$\mathbf{S}_{\mathbf{xx}}(\omega) = \mathbf{H}(\omega)\mathbf{S}_{\mathbf{ff}}(\omega)\mathbf{H}^{H}(\omega), \tag{5}$$

where $\mathbf{S}_{\mathbf{ff}}(\omega)$ is the PSD of the input random process; $\mathbf{H}(\omega)$ is the Fourier transform of the impulse response function given by (4); and $(\cdot)^H$ denotes complex conjugate transpose of a matrix. The desired output can be extracted from $\mathbf{S}_{\mathbf{xx}}(\omega)$ using an output matrix, $\mathbf{C}$, as:

$$\mathbf{y}(\omega) = \mathbf{C}\mathbf{S}_{\mathbf{xx}}(\omega)\mathbf{C}^{\mathrm{T}}. \tag{6}$$

The model, $\mathcal{M}$, in such problems is the physical system, represented using a frequency response matrix; the input is the PSD of an externally applied load; and the relevant output is the PSD of the desired response of the system.

2.3. Static deflection of a mechanical system

The static analysis of a system subjected to a time-invariant load can be performed by first spatially discretizing the system to formulate its stiffness matrix, $\mathbf{K}$, and then computing its static deflection as:

$$\mathbf{u} = \mathbf{K}^{-1}\mathbf{f}, \tag{7}$$

where $\mathbf{u}$ is the static response of the system and $\mathbf{f}$ is a vector of the static load as applied to the discretized model of the system. The desired output can be computed from $\mathbf{u}$ using an output matrix, $\mathbf{C}$, as:

$$\mathbf{y} = \mathbf{C}\mathbf{u}. \tag{8}$$

The model, $\mathcal{M}$, in such problems is the physical system, represented using a system of linear algebraic equations; the input is an externally applied load; and the output is the response of the desired parts of the system.

2.4. Numerical quadrature

Numerical quadrature finds many applications in uncertainty quantification problems, especially when aleatoric uncertainties are involved. The most common application is the numerical approximation of the expectation operator. The expected value of a function,

$g(\mathbf{q})$, of N random variables, $\mathbf{q} = \{q_1, \ldots, q_N\}^{\mathrm{T}}$, having a joint probability density function, $f_{\mathbf{Q}}(\mathbf{q})$, can be computed as:

$$E[g(\mathbf{q})] = \underbrace{\int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty}}_{N} g(q_1, \ldots, q_N) \times f_{\mathbf{Q}}(q_1, \ldots, q_N)\, \mathrm{d}\mathbf{q}. \tag{9}$$

Although analytical integration of (9) can be performed for certain types of distributions, e.g. the Gaussian distribution, numerical quadrature is unavoidable in most scenarios. In such cases, the multi-dimensional integration in (9) can be computed approximately using a quadrature rule as:

$$E[g(\mathbf{q})] \approx \underbrace{\sum_{i_1=1}^{N_{\mathrm{a}}} \cdots \sum_{i_N=1}^{N_{\mathrm{a}}}}_{N} w_{i_1,\ldots,i_N} g(q_{i_1}, \ldots, q_{i_N}) \times f_{\mathbf{Q}}(q_{i_1}, \ldots, q_{i_N}), \tag{10}$$

where N_{a} is the number of abscissas used by the quadrature rule and $w_{i_1,\ldots,i_N}$ are the associated weights. Since the quadrature rule in (10) is constructed as a tensor-product of a one-dimensional quadrature rule, Gauss–Hermite quadrature in this case, the abscissas and the weights in (10) can be computed using appropriate combinations of the one-dimensional abscissas and weights. Due to the use of a full tensor-product grid, the computational complexity of the multi-dimensional numerical quadrature in (10) increases exponentially with the number of dimensions; in particular, the number of quadrature points in the N-dimensional space is $N^{N_{\mathrm{a}}}$. Thus, a multi-dimensional numerical quadrature can be computationally very demanding.

The model, $\mathcal{M}$, in such problems is the integration operator of appropriate dimensions; the input is the function of random variables; and the output is the value of the integration.

3. Numerical examples

This section investigates the computational efficiency of each of the methods introduced in the previous section. The computational times required by GPU implementations are compared to that of corresponding single- and multi-core CPU implementations for each method. One NVIDIA Tesla C1060 GPU and two Intel Xeon E5530 quad-core CPUs have been utilized to perform all of the timing studies. Moreover, it is well known that GPUs typically perform better in single precision (SP) compared to double precision (DP) arithmetic because of their hardware structure. For instance, the NVIDIA Tesla GPU has a peak performance of 933 Gflop/s for SP compared to just 125 Gflop/s for DP. Therefore, in all the examples, timing comparisons have been performed for both SP and DP computations. Since there is a negligible performance gap in SP and DP computations performed on a CPU, such a distinction has not been made for the multi-core CPU computational gains.

Let t_{SC}, t_{MC} and t_{GP} denote the computational times for the execution of the single-core CPU, multi-core CPU and GPU implementations of an algorithm respectively. Gains in computational efficiency can been defined as:

$$S_{\mathrm{MC}} = \frac{t_{\mathrm{SC}}}{t_{\mathrm{MC}}}, \qquad S_{\mathrm{GS}} = \frac{t_{\mathrm{SC}}}{t_{\mathrm{GP}}} \quad \text{and} \quad S_{\mathrm{GM}} = \frac{t_{\mathrm{MC}}}{t_{\mathrm{GP}}}. \tag{11}$$

Thus, S_{MC} is the gain in computational efficiency obtained by the multi-core CPU implementation compared to the single-core CPU implementation, and S_{GS} and S_{GM} are the gains in computational efficiency obtained by the GPU implementation compared to the single- and multi-core CPU implementations, respectively.

The performance of the parallel implementation has been assumed to be optimal if the observed gain in computational efficiency is close to its theoretical maximum. This theoretical prediction is straightforward for the multi-core CPU implementations; since eight computational cores are available on the CPU, the optimal multi-core CPU performance gain is expected to be eight times that of the single-core CPU performance. Establishing a theoretical peak for performance gain for the GPU implementation is highly problem dependent due to hardware constraints which govern the optimization of the GPU kernel for performance. The optimality of the GPU implementation has been discussed in each of the examples individually. The discussion of the GPU kernel-optimization utilizes the terms memory-only performance and compute to global memory access (CGMA) ratio.

Memory-only performance is defined as the global-memory throughput obtained when only memory read/write operations are performed, and all the computations are omitted. This is an important metric to indicate whether the global memory accesses are coalesced. The closer the observed memory throughput is to the theoretical peak memory bandwidth (102 GB/s in this study), the better.

CGMA ratio is defined as the ratio of the total number of computations to the total number of global memory accesses. This ratio indicates whether the GPU implementation is compute- or memory-bound. When compute-bound, the reference peak performance of the GPU implementation is governed by the theoretical peak Gflop/s (933 Gflop/s for SP, 125 Gflop/s for DP in this study). When memory-bound, the peak theoretical performance can be computed involving the peak theoretical bandwidth as:

$$\text{Peak theoretical performance} = \text{CGMA} \cdot \frac{\text{Peak theoretical bandwidth}}{B}, \quad (12)$$

where B is the number of bytes required to store the data ($B = 4$ for SP, $B = 8$ for DP). Thus, the theoretical peak performance of the GPU implementation presented in this study will be memory-bound if

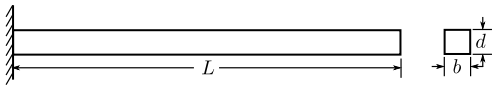
$$\text{CGMA} < \begin{cases} 36.6 & \text{for SP,} \\ 9.8 & \text{for DP,} \end{cases} \quad (13)$$

otherwise, it will be compute-bound.

The CUDA visual profiler has been utilized to obtain the memory-throughput of the GPU implementations while the CGMA ratio has been computed manually. Moreover, the largest available problem-size considered has been utilized to estimate the memory-throughput and the computational performance of the algorithms.

3.1. Example 1

The first example considers a cantilevered Eüler–Bernoulli beam, as shown in Fig. 5. The beam is subjected to a random, static tip load. A finite element discretization with cubic shape functions has been utilized to obtain the stiffness matrix, **K**, and the load vector, **f**. The static deflection of the beam is computed using (7) due to 1000 different samples of the load. The `sgetrf` and `dgetrf` routines have been utilized from the Intel's MKL library [12], and their inherent parallelism



$E = 2.068 \times 10^8$ N/m^2 $\quad L = 9.144$ m
$b = d = 0.0762$ m $\quad \rho = 8140$ kg/m^3

Fig. 5. Cantilever beam model.

Table 2
Gains in computational efficiency for Example 1

No. of DOF	S_{MC}	S_{GS}		S_{GM}	
		SP	DP	SP	DP
128	1.00	1.95	2.00	1.95	2.00
512	1.56	3.67	6.25	2.35	4.01
1024	3.88	2.25	2.38	0.58	0.61
2048	6.31	4.04	3.56	0.64	0.56
4096	6.06	5.97	4.19	0.99	0.69
8192	6.46	32.10	27.65	4.97	4.28
16,384	7.31	213.85	205.33	29.25	28.09

has been utilized for the multi-core CPU implementation. The corresponding routines from the MAGMA [14] library have been utilized for the GPU implementation; both have been assumed to be optimal in performance based on the evidence available in their respective documentations. This example utilizes both Type I and Type II parallelization strategies.

Table 2 shows the gains in computational efficiency obtained for this example. The GPU implementation shows gains in computational efficiency of approximately 200 times compared to the single-core CPU implementation and approximately 30 times compared to the multi-core CPU implementation for both SP and DP computations for the largest considered problem size. This example exclusively uses existing GPU library routines.

3.2. Example 2

The second example considers the same beam model as in the previous example. It is now subjected to a random dynamic load, governed by a first-order autoregressive, stationary random process, prescribed in the form of its PSD, and the analysis is performed in the frequency domain. The PSD of the response of all of the states is directly computed using (5). Since the input is present in only one degree-of-freedom of the system, (5) can be computed by employing a rank-1 update algorithm followed by scaling with a complex scalar. The rank-1 update has been performed using the `cherk` and `zherk` BLAS routines from the In-

Table 3
Gains in computational efficiency for Example 2

No. of DOF	S_{MC}	S_{GS}		S_{GM}	
		SP	DP	SP	DP
100	0.32	0.91	0.48	1.02	1.50
500	0.97	14.71	6.09	15.68	6.30
1000	1.31	34.14	14.16	35.27	10.80
5000	1.32	68.17	27.05	66.76	20.42
7500	1.32	71.78	28.47	67.97	21.49
10,000	1.32	73.82	29.29	71.17	22.11

tel's MKL library for the CPU implementation. The scaling with the complex scalar has been performed using a vectorized loop obtained by aggressive optimization of the code via the compiler. Inherent parallelism of the MKL BLAS routines has been exploited to optimize the multi-core CPU performance. On the GPU, the rank-1 update is performed using the corresponding BLAS routines from the CUBLAS [19] library. A new GPU kernel has been developed for the scaling. Since there is no need for sampling, this example utilizes only the Type II parallelization strategy.

The gains in the computational efficiency obtained for this example are shown in Table 3. The multi-core CPU implementation does not show much gain in computational efficiency because of the relatively small size of the matrices involved. The GPU implementation, however, shows about 70 times better performance for SP, and about 20 times better performance for DP computations for the largest considered problem size.

About 98% of the computational time of the GPU implementation is consumed by the BLAS routines which have been assumed to be already optimized for performance. The superior computational performance of CUBLAS routines when compared to corresponding MKL BLAS routines may be attributed to:

(i) efficient memory access of 2D data structures via the use of texture memory maps on GPUs,
(ii) increased scalability of algorithms designed to use GPUs,
(iii) simple thread control logic of GPUs, and/or
(iv) ability of GPUs to hide memory latency by constantly switching between active and inactive thread warps.

The new GPU kernel, which scales the involved matrix–matrix product to obtain the spectral response of the system, is memory-bound, with a CGMA ratio of 2.25 and has been assumed optimal in performance because it shows a computational performance of 50.67 Gflop/s in SP and 23.82 Gflop/s in DP against the theoretically predicted performances of 57.38 and 28.69 Gflop/s, respectively. This kernel utilizes n^2 threads, n being the number of degrees-of-freedom of the system, i.e., each thread scales one element of the involved matrix. The real and imaginary components of each element are stored in contiguous memory locations by utilizing the `cuComplex` and `cuDoubleComplex` data structures provided by the CUDA runtime library. In order to reduce the number of memory accesses, each thread reads a matrix element into its register memory, computes the complex product, and writes the scaled element back to the global memory. The complex scalar by which each matrix element is to be multiplied is stored in the constant memory space which allows for fast concurrent access of this constant by all the threads and also saves two register variables from being used by each thread.

This example demonstrates the mixed approach, where existing GPU library routines have been utilized in conjunction with a custom GPU kernel. However, the majority of the computation is still being performed by the vendor libraries.

3.3. Example 3

The third example considers the same beam model as in the previous two examples but now subjected to a time-dependent random load, governed by a first-order autoregressive, stationary random process. The equations of motion for this example are written as in (2), and the response is computed using the convolution integral, as given by (3). 100 samples of the response were computed and the output matrix, $\mathbf{C}$, was assumed to be an identity matrix, i.e., all the states were computed.

The convolution has been performed using the FFT, which requires three steps: (i) computing the FFT of the two vectors, (ii) computing the element-by-element product of the transformed vectors and (iii) computing the inverse FFT of the product. The FFTW [8] and the CUFFT [20] libraries have been utilized to perform the FFT and the inverse FFT on CPU and GPU, respectively. The built-in shared-memory parallelism of the FFTW library has been utilized along with a pthread implementation of the element-by-element multiplication to obtain the multi-core CPU implementation. A custom kernel has been implemented to perform the element-by-element product on the GPU. Both Type I and Type II parallelization strategies have been employed in this example.

Table 4
Gains in computational efficiency for Example 3

No. of DOF	S_{MC}	S_{GS}		S_{GM}	
		SP	DP	SP	DP
		$N_t = 1024$			
60	6.23	245.44	132.54	45.11	21.26
124	6.84	233.62	145.32	37.49	21.26
252	7.33	294.83	151.90	44.01	20.73
		$N_t = 8192$			
64	7.36	337.54	384.71	54.52	52.28
128	7.42	692.61	806.69	102.65	108.69
256	7.40	1206.95	881.04	181.84	119.05

The gain in computational efficiency obtained using the multi-core CPU implementation approaches 8 and hence, has been characterized to be optimal. Table 4 shows the gains in computational efficiency obtained by the GPU implementation for different cases. For the largest considered problem size, the GPU implementation shows a gain in computational efficiency of about 1200 and 180 times for SP and DP computations, respectively, compared to the single-core CPU implementation, and about 880 and 120 times compared to the multi-core CPU implementation for SP and DP computations, respectively.

In the GPU implementation, the FFT operations only consume about 20% of the computational time. Since the CUFFT library has been utilized, this portion of computation is assumed to be already optimal in performance. The new GPU kernel, which performs the element-by-element multiplication of the pertinent complex matrices, has a memory-only performance of 80.57 GB/s. With a CGMA ratio of 1.5 for both SP and DP computations, it is clearly memory-bound, having a theoretical peak performance of 38.25 Gflop/s in SP and 19.12 Gflop/s in DP computations, as computed from (12). In the new GPU kernel, one thread-block computes one sample of the response of the system. The observed performance is 26.50 and 15.75 Gflop/s in SP and DP respectively, which is approximately 70% and 82% of the theoretically predicted performance for SP and DP. Such behavior is expected because the custom GPU kernel is memory-bound, and shows a memory throughput of approximately 79% of the theoretically available memory-bandwidth. The custom kernel again utilizes the `cuComplex` and `cuDoubleComplex` data structures provided by the CUDA runtime library as in the previous example. The transformed impulse response matrix, **H**, of the system is stored as a two-dimensional (2D) array of size $N_f \times N_s$ where N_f is the number of frequency points and N_s is the number of states squared. Samples of the load, **P**, are stored in a 2D array of size $N_f \times N_{ls}$ where N_{ls} is the number of load samples. Each thread-block computes the element-by-element product of **H** with one column of **P**. Within a thread-block, each thread performs element-by-element product of one row of **H** with the corresponding element of its load vector. Such a task division provides two advantages: (i) since the matrices are stored in column-major format, at a given time successive threads access successive elements of a column of the matrix which ensures memory coalescence and (ii) each thread-block can collectively store the required sample of the load vector in shared-memory which reduces the number of global memory accesses.

This example again demonstrates the mixed approach, where existing GPU library routines have been utilized in conjunction with a new GPU kernel. However, in this example, the majority of the computation is performed by the new kernel, and consequently, the gains in computational efficiency are considerably higher when compared to those of the previous two examples.

3.4. Example 4

The fourth example considers a single degree-of-freedom dynamical system as shown in Fig. 6. The system is assumed to have a nonlinear damping term and is subjected to a random load given by:

$$f(t) = A \cdot [\sin(t) + \varepsilon_t], \tag{14}$$

where A is the amplitude of the load and ε_t is a unit intensity Gaussian white noise random process. The equation of motion of the dynamical system is given by:

$$m\ddot{x} + c\dot{x} + \text{sgn}(\dot{x})|\dot{x}|^{0.5} + kx = f(t), \tag{15}$$

where $m = 2$ kg, $c = 1.6$ Ns/m and $k = 62.84$ N/m are the mass, damping and stiffness of the system, respectively. A fourth-order explicit Rünge–Kutta method has been employed to numerically integrate the equa-

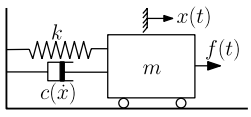


Fig. 6. Single degree-of-freedom system.

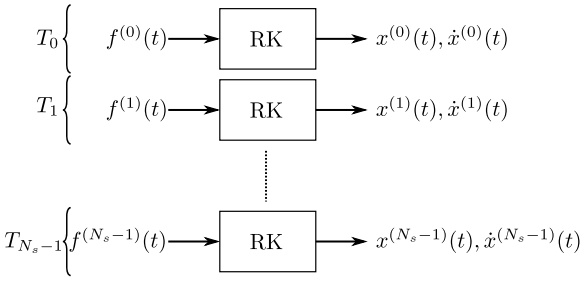


Fig. 7. Work division between threads for Example 4.

tion of motion. A 10 s time-history has been computed with $N_t = 1001$ discrete points in time for a varying number of samples, N_s, of the input.

Since the size of the system is small, parallelism is exploited across the number of samples, i.e., Type I parallelization strategy has been employed. In both the GPU and the CPU implementations, each thread computes one sample of the response of the system, as shown in Fig. 7. 'RK' denotes the Rünge–Kutta ODE solver, $f^{(i)}(t)$ is the ith sample of the random input to the system and $x^{(i)}(t)$ and $\dot{x}^{(i)}(t)$ are the corresponding displacement and velocity time-histories. T_i represents ith parallel thread, i.e., thread with index i. A pthread implementation has been utilized to distribute the work among parallel CPU threads. Table 5 shows the gains in computational efficiency obtained for this example, given as a function of the number of samples computed. The gain in computational efficiency obtained using the multi-core CPU implementation asymptotically approaches 8, which is the maximum achievable theoretical value, as discussed earlier. The GPU implementation displays a gain in computational efficiency of about 2300 times compared to the single-core CPU implementation and about 300 times compared to the multi-core CPU implementation for SP computations for the largest considered problem size. The corresponding gains in computational efficiency for DP computations are observed to be 140 and 19 times, respectively.

The CGMA ratio is 49 for SP and 72 for DP. Since the number of global memory accesses is the same for both SP and DP, the difference in the CGMA is due to the difference in the number of compute instructions. Though the same GPU kernel is utilized for both SP and DP computations, the difference in the number of compute instructions arises due to different implementations of the divide and the sine function at compiler-level. CUDA implements fast versions of floating-point division and transcendental functions (sine, cosine, etc.) for SP, which are not available for

Table 5
Optimized gains in computational efficiency for Example 4

No. of samples	S_MC	S_GS		S_GM	
		SP	DP	SP	DP
1	0.39	0.48	0.07	1.25	0.20
10	1.88	4.63	0.73	2.47	0.39
100	2.56	44.16	5.22	17.24	2.06
1000	4.39	426.93	51.80	97.27	11.94
10,000	6.58	1994.32	121.78	303.09	18.83
50,000	7.47	2321.44	135.79	310.72	18.57
100,000	7.63	2336.58	138.08	306.37	18.37

DP, thus, causing the performance gap observed between the two. Based on the CGMA ratio, it is clear that the GPU implementation is compute-bound.

The observed performance for SP and DP computations is 536.78 and 51.48 Gflop/s, respectively. These are approximately 50% of that of their respective theoretical peaks. The difference between the observed and the theoretical performance of the GPU implementation can be attributed to low processor occupancy (25% for SP and 18.8% for DP computations) and high instruction overhead. The processor occupancy is low due to high register pressure which, again, is due to a fairly large number of instructions in the GPU kernel. The size of the GPU kernel is large because each thread is used to compute one sample of the dynamic response of the system, and therefore, each thread must implement the four-stage explicit Rünge–Kutta method to solve the dynamical system. All the loops have been unrolled inside the kernel in order to improve computational efficiency, which also adds to the size of the kernel.

A non-conventional data-structure has been utilized for storing the samples of the dynamic response of the system in order to ensure that the accesses to the global memory are coalesced. An intuitive data-structure to store the response samples of the system, which has been utilized for the single- and multi-core CPU implementations is shown in Fig. 8. The subscripts denote the time-step and the superscripts denote the sample number of the response, i.e., $x_i^{(j)} = x^{(j)}(t_i)$ is the jth sample of the displacement at time t_i of the system. Adjacent cells represent contiguous memory locations.

It can be observed that each thread accesses two sets of contiguous memory locations in order to store the displacement and the velocity of the system, which are separated by N_t memory locations. As the time-marching progresses, these two memory locations shift to the right by a stride of unity. Threads with consecu-

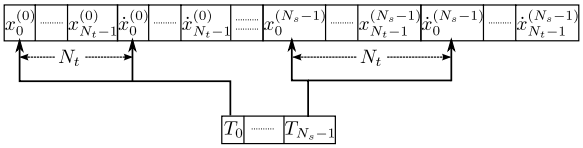


Fig. 8. Intuitive data-structure used for CPU implementations of Example 4.

tive indices access memory locations that are separated by a distance $2N_t$, which is the number of memory locations required to store entirely, one sample of the dynamic response of the system. Thus, adjacent threads do not access adjacent memory locations. Such a memory access pattern does not affect the performance of the CPU implementations. However, in GPU implementations, such a memory access pattern is referred to as non-coalesced memory access and can cause considerable degradation in the performance of the GPU kernel.

If a similar data-structure is utilized for the GPU implementation, its memory-only performance is only 4.68 GB/s, which is considerably less than the theoretically attainable peak memory throughput of 102 GB/s. Consequently, the observed gains in computational efficiency are considerably reduced as shown in Table 6. It can be observed that the efficiency for SP is diminished by an order of magnitude. If the kernel was memory-bound with a memory throughput of 4.68 GB/s, the performance for SP and DP computations would be 28.66 and 42.12 Gflop/s, respectively, as compared to 536.78 and 51.48 Gflop/s observed for the optimized kernel. This is why the performance of SP computations is more severely affected than that of DP computations. Moreover, it can be observed that a non-coalesced memory access pattern can render an otherwise compute-bound kernel, memory-bound thus drastically degrading its performance.

In order to ensure coalesced memory access and improve the memory throughput, a non-conventional data-structure has been utilized for the GPU implementation, as shown in Fig. 9. Each thread accesses two contiguous memory locations to store its sample of the displacement and the velocity of the system for current time; successive threads utilize successive memory locations to store their respective samples. As the time-marching progresses, the entire access pattern is shifted towards the right by a stride of $2N_\mathrm{s}$. Thus, memory coalescing is ensured for the entire time-marching scheme. The memory-only performance of the optimized GPU implementation is 76.4 GB/s, which is considerably better compared to the earlier 4.68 GB/s.

Table 6
Unoptimized gains in computational efficiency for Example 4

No. of samples	S_{MC}	S_{GS}		S_{GM}	
		SP	DP	SP	DP
1	0.29	0.34	0.07	1.17	0.24
10	1.82	1.26	0.71	0.69	0.39
100	2.26	9.37	5.33	4.15	2.34
1000	4.49	94.03	51.20	20.94	11.40
10,000	6.54	142.60	111.60	21.80	17.06
50,000	7.07	145.74	120.69	20.61	17.07
100,000	7.33	147.72	122.35	20.15	16.69

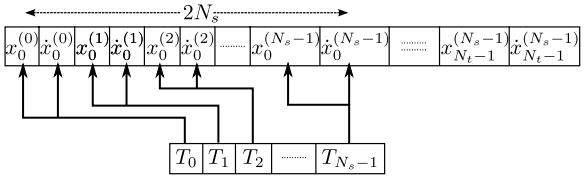


Fig. 9. Data-structure used for GPU implementation of Example 4.

This example demonstrates the second implementation approach where a completely new GPU kernel has been implemented and demonstrates the best gains in computational efficiency among all the examples presented herein.

3.5. Example 5

In the final example, Gauss–Hermite quadrature has been utilized to compute the expected value of a function of N standard, normally distributed random variables. The function to be integrated is given by:

$$g(\mathbf{q}) = \mathbf{q}^{\mathrm{T}}\mathbf{q}. \tag{16}$$

Both the CPU and the GPU implementations first evaluate the integrand at the quadrature points in parallel, and then perform a parallel sum-reduction to compute the approximate expected value. OpenMP [24] has been utilized for the multi-core CPU implementation. Four Gauss points ($N_\mathrm{a} = 4$) in each dimension were utilized for the quadrature.

Table 7 shows the gains in computational efficiency obtained for this example. For the largest considered problem size, the GPU implementation shows gains in computational efficiency of about 50 times and 30 times for SP and DP computations, respectively, when compared to the single-core CPU implementation. Compared to the multi-core CPU implementation, the corresponding gains in computational efficiency

Table 7
Gains in computational efficiency for Example 5

No. of dimensions	S_{MC}	S_{GS}		S_{GM}	
		SP	DP	SP	DP
6	0.01	1.17	1.34	403.89	149.89
7	0.02	3.42	4.13	368.69	259.41
8	0.04	15.01	8.60	345.59	204.16
9	0.16	33.76	25.07	116.11	153.53
10	1.71	39.31	37.02	19.43	21.70
11	2.74	49.44	34.00	19.23	12.41
12	4.46	56.12	27.31	14.71	6.13
13	4.75	56.59	28.90	11.76	6.08
14	5.92	54.48	27.98	10.68	4.72
15	6.16	54.43	28.11	9.60	4.56

are about 10 and 5 times, respectively for SP and DP computations. While the gains in computational efficiency shown by the GPU implementation compared to multi-core CPU implementation is not as great as those of the previous examples, it should be noted that for smaller dimensions, the multi-core CPU implementation is computationally much inferior to the single-core CPU implementation while the GPU implementation offers reasonable efficiency gains.

The GPU implementation consists of two kernels. The first kernel evaluates the integrand at the quadrature points and performs a parallel sum-reduction on the data to compute the expected value. However, the parallel reduction can only be performed within a thread-block due to the fact that threads from different thread-blocks cannot communicate with each other. Therefore, a second GPU kernel is required, which performs the parallel sum-reduction on the data recursively until the data can fit within one thread-block and the approximate expected value can be computed.

The first kernel consumes about 99% of the total computational time and thus governs the performance of the GPU implementation. It achieves a memory throughput of only 0.83 GB/s due to conditional access of the global memory by the threads. The CGMA ratio changes with the number of dimensions in the first kernel because as the number of dimensions increase, the number of arithmetic operations increase considerably while the number of global memory accesses increase only slightly.

Each thread is utilized to compute the value of the integrand at one mesh-point. Since a tensor-product multi-dimensional quadrature is being performed, the abscissas and the weights at each mesh-point are computed as the combination of appropriate abscissas and weights of the corresponding one-dimensional quadrature rule. In order to avoid conditional statements based on the thread-index to determine which abscissas and weights are to be utilized for each function evaluation, the thread-indices are converted into length-N, base-N_{a} bit-strings, using the first few required ASCII characters. These bit-values can be directly cast into integers that index the abscissas and the weights of the one-dimensional quadrature rule required to compute the abscissa and the weight at a particular mesh-point of the N-dimensional space. The function values, now being computed, are stored in a shared-memory array within each thread-block, which is utilized towards the end of the first kernel to perform a preliminary parallel sum-reduction.

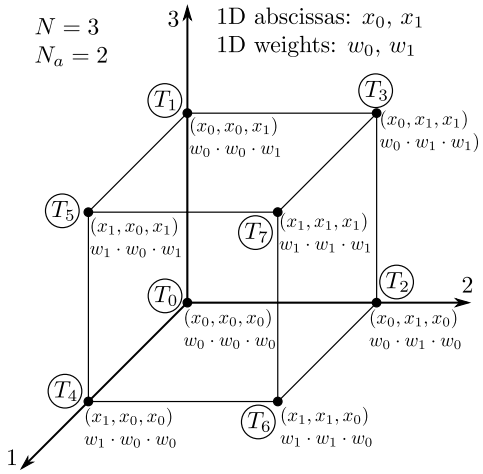


Fig. 10. Mesh decomposition into parallel threads for Example 5.

This idea is illustrated in Fig. 10. A mesh is shown with $N = 3$ and $N_{\text{a}} = 2$. The tensor-product abscissas and the corresponding weights are shown near each mesh-point. Thread index is also shown in Fig. 10; this index is utilized to evaluate the function at that particular mesh-point. The mesh-points are assigned to the threads in a way that a length-3, base-2 bit-string of the thread index can be utilized to determine the combination of the one-dimensional quadrature rule abscissas and weights, required for the tensor-product quadrature rule. It can be seen that these bit-strings, $0_{10} = (000)_2$, $1_{10} = (001)_2$, $2_{10} = (010)_2$, etc., represent the combination of the abscissas and the weights exactly, and conditional statements can be avoided.

Although the CGMA ratio increases with the number of dimensions, 20 for 6 dimensions to 47 for 15 dimensions, rendering the implementation compute-bound for larger dimensions, the observed performance is only 50.61 Gflop/s for SP and 24.26 Gflop/s for DP computations. There are two primary sources

of the sub-optimal performance of the GPU implementation of this example: (i) the presence of a large number of thread synchronization points within the thread-blocks and (ii) the presence of conditional statements causing divergent branches of threads. Both the synchronization points and the conditional statements are necessitated by the parallel sum-reduction.

This example also utilizes a completely new GPU implementation, but the gains in computational performance are not as great as in the previous example due to the different nature of the problem, as discussed above.

4. Discussion and conclusions

GPUs have rapidly expanded their influence to general purpose computing problems and have established themselves as competent alternatives to the traditional CPU-based parallel computing. They offer teraflop/s computing power within the energy and financial demands of that of a usual desktop computer. A large number of existing algorithms have already been implemented on GPU to harness its enormous computing power and as a result, more and more researchers are exploring the potential of GPU computing in their respective fields.

This study presented GPU implementations of five techniques frequently used in the uncertainty quantification of dynamical and mechanical systems and provided significant evidence of the potential of the use of GPUs in this area. The first four examples presented implementations of static and dynamic reanalysis techniques, which can be easily adapted to other analysis procedures, such as sensitivity analysis, model validation and calibration, and design exploration and optimization. The final example presented the implementation of a multi-dimensional numerical quadrature scheme, which is utilized in many uncertainty quantification procedures. These procedures include non-intrusive methods, such as sampling strategies, deterministic integration approach and sparse grid cubature, as well as intrusive methods, such as the stochastic Galerkin method.

It is expected that the GPU implementations presented in this study will serve as initial bases for further developments in the use of GPUs in the field of uncertainty quantification. The specific discussions regarding the GPU kernels developed for the numerical examples are expected to (i) aid the understanding of the performance constraints on the relevant GPU kernels and (ii) provide some guidance regarding the computational and the data structures to be utilized in novel applications of GPU computing in the problems related to uncertainty quantification.

The performance of the new algorithms was measured and compared with corresponding single- and multi-core CPU implementations and gains in computational efficiency of up to three orders of magnitude with respect to single-core CPU implementations and two orders of magnitude with respect to multi-core CPU implementations were observed. Thus, a single GPU showed computational performance equivalent to a cluster of CPUs consisting of approximately $\mathcal{O}(10^2–10^3)$ cores for the analyses considered.

The gains in efficiency presented here promise to be further increased as the NVIDIA Tesla GPU utilized for this study has been superseded by the newer Fermi GPU from NVIDIA. The Fermi offers considerable upgrades in performance compared to the Tesla. It has a larger number of computational cores, larger global and shared memory, a greater number of registers per thread, higher memory bandwidth, and greatly increased double precision performance. More importantly, the shared memory of the Fermi can also be utilized as L1 cache at the discretion of the programmer which may help hide the memory latency to a great extent. The larger number of available registers per thread along with the increased double precision performance will further increase the computational performance of Example 4, which is compute-bound and performs poorly in double precision due to register pressure. The increased memory bandwidth and the availability of L1 cache will have a direct impact on the performance of the remaining examples which are memory-bound. The availability of a greater number of computational cores will have a positive effect on the efficiency of all the examples as more thread warps can be executed concurrently. In addition, the Fermi also has error code checking (ECC) support which improves the performance of clusters of GPUs.

The hardware development of the GPUs and their application in general purpose computing have formed a complimentary pair with each of them driving the development of the other. This trend can be clearly observed from the hardware evolution history of the GPUs along with the footprint of the applications that utilize the computing power of the GPUs. It has been shown in this study that GPUs offer considerable advantages in uncertainty quantification problems in computational mechanics and dynamics. While an exclusive use of GPUs may not be optimal for very large

problems, there is definitely potential for the mixed use of CPUs and GPUs in this area. This use of heterogeneous computing resources in uncertainty quantification is one thrust of future investigations in this area.

Acknowledgement

Gaurav gratefully acknowledges the support of this work by a University of Minnesota Doctoral Dissertation Fellowship.

References

[1] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams and K.A. Yelick, The landscape of parallel computing research: a view from Berkeley, Technical Report USB/EECS-2006-183, University of California at Berkeley, 2006.

[2] N. Bell and M. Garland, Efficient sparse matrix-vector multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, 2008.

[3] J. Bolz, I. Farmer, E. Grinspun and P. Schröoder, Sparse matrix solvers on the GPU: conjugate gradients and multigrid, in: *ACM SIGGRAPH 2003 Papers*, ACM, New York, NY, USA, 2003, pp. 917–924.

[4] A.R. Brodtkorb, C. Dyken, T.R. Hagen, J.M. Hjelmervik and O.O. Storaasli, State-of-the-art in heterogeneous computing, *Scientific Programming* **18**(1) (2010), 1–33.

[5] J. Cohen and M. Garland, Novel architectures: solving computational problems with GPU computing, *Computing in Science and Engineering* **11**(5) (2009), 58–63.

[6] P. Colella, Defining software requirements for scientific computing, DARPA HPCS Presentation, 2004.

[7] H. Contreras, The stochastic finite-element method, *Computers and Structures* **12**(3) (1980), 341–348.

[8] Fastest Fourier Transform in the West (FFTW), http://www.fftw.org/, 2010.

[9] GPGPU, Gpgpu, http://gpgpu.org, 2010.

[10] J.M. Hammersley and D.C. Handscomb, *Monte Carlo Methods*, Wiley, New York, 1964.

[11] Intel, Intel Xeon quad-core e5530, http://ark.intel.com/Product.aspx?id=37103, 2010.

[12] Intel, Math Kernel Library, http://software.intel.com/en-us/intel-mkl/, 2010.

[13] M. Januszewski and M. Kostur, Accelerating numerical solution of stochastic differential equations with CUDA, *Computer Physics Communications* **181** (2009), 183–188.

[14] MAGMA, Matrix Algebra on GPU and Multicore Architectures, http://icl.cs.utk.edu/magma/index.html, 2010.

[15] M.D. McKay, W.J. Conover and R.J. Beckman, A comparison of three methods for selecting values of input variables in the analysis of output from a computer code, *Technometrics* **21**(2) (1979), 239–245.

[16] G.E. Moore, Cramming more components onto integrated circuits, *Electronics* **38**(8) (1965), 114–117.

[17] J. Neyman, On the two different aspects of the representative method: the method of stratified sampling and the method of purposive selection, *Journal of the Royal Statistical Society* **97**(4) (1934), 558–625.

[18] J. Nickolls and W.J. Dally, The GPU computing era, *IEEE Micro* **30**(2) (2010), 56–69.

[19] NVIDIA, CUBLAS User's Manual, v3.1, 2010.

[20] NVIDIA, CUFFT User's Manual, v3.1, 2010.

[21] NVIDIA, NVIDIA CUDA zone, http://www.nvidia.com/object/cuda_apps_flash_new.html, 2010.

[22] NVIDIA, NVIDIA Programming Guide, v3.1.1, 2010.

[23] NVIDIA, NVIDIA Tesla C1060, http://www.nvidia.com/object/product_tesla_c1060_us.html, 2010.

[24] OpenMP, http://www.openmp.org, 2010.

[25] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone and J.C. Phillips, GPU computing, *Proceedings of the IEEE* **96**(5) (2008), 879–899.

[26] J.D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn and T.J. Purcell, A survey of general-purpose computation on graphics hardware, *Computer Graphics Forum* **26**(1) (2007), 80–113.

[27] P.P.M. So, Time-domain computational electromagnetics algorithms for GPU based computers, in: *International Conference on Computer as a Tool (EUROCON'2007)*, Warsaw, IEEE Computer Society, Washington, DC, USA, 2007, pp. 1–4.

[28] X. Tian and K. Benkrid, Mersenne twister random number generation on FPGA, CPU and GPU, in: *NASA/ESA Conference on Adaptive Hardware and Systems (AHS'2009)*, San Francisco, CA, IEEE Computer Society, Washington, DC, USA, 2009, pp. 460–464.

[29] S. Tomov, J. Dongarra and M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Computer Science Technical Report UT-CS-08-632 (also LAPACK working note 210), University of Tennessee, 2008.

[30] S. Tomov, M. McGuigan, R. Bennett, G. Smith and J. Spiletic, Benchmarking and implementation of probability-based simulations on programmable graphics cards, *Computers and Graphics* **29**(1) (2005), 71–80.

[31] F. Vázquez, E.M. Garzón, A. Martínez and J.J. Fernández, The sparse matrix vector product on GPUs, Technical Report CSTN-077, University of Almeria, 2009.

[32] S.D.C. Walsh, M.O. Saar, P. Bailey and D.J. Lilja, Accelerating geoscience and engineering system simulations on graphics hardware, *Computer and Geosciences* **35** (2009), 2353–2354.

[33] Y. Zhao, Lattice Boltzmann based PDE solver on the GPU, *The Visual Computer* **24**(5) (2008), 323–333.